Type Evolution and Instance Adaptation

Stewart M. Clamen

June 1992
CMU-CS-92-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

Schema evolution support is an important facility for object-oriented database (OODB) systems. While existing OODB systems provide for limited forms of evolution, including modification to the database schema and reorganization of affected instances, we find their support insufficient. Specific deficiencies are 1) the lack of compatibility support for old applications, and 2) the lack of ability to install arbitrary changes upon the schema and database.
This paper examines the limitations of existing schemes, and offers a more general framework for specifying and reasoning about the evolution of class definitions and the adaptation of existing, persistent instances to those new definitions.

This research was sponsored in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597; and by the Office of Naval Research under Contract N00014-88-K-0641. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Schema evolution, object-oriented databases, class versioning, instance adaptation, compatibility

Introduction

Database systems exist to support the long-term persistence of data. It is natural to expect that, over time, needs will change and that those changes will necessitate a modification to the interface for the persistent data. In an object-oriented database (OODB) system, such a situation would motivate an evolution of the database schema. For this reason, support for **schema evolution** is a required facility in any serious OODB system.

An OODB database schema consists of class definitions and an inheritance hierarchy. A class is a tuple of methods and attributes. The database is populated by instances of those classes, with values for each of the attributes. The schema describes the interface between the set of application programs and the persistent repository of objects. When the schema changes so does the interface, possibly leaving incompatible elements on both sides. We are interested in the problem of managing existing database objects, what we call the **instance adaptation problem**. This paper examines the limitations of existing adaptation schemes and offers a more general framework for specifying and reasoning about the evolution of class definitions and the adaptation of existing, persistent instances to those new definitions.

Two general instance adaptation strategies have been identified and implemented by various OODB systems. The first strategy, **conversion**, restructures the affected instances to conform to the representation of their modified classes. Conversion is supported by the ORION [2, 16] and GemStone [6] systems.

The primary shortcoming of the conversion approach is its lack of support for **program compatibility**. By discarding the former schema, application programs that formerly interacted with the database through the changed parts of the interface are now obsolete. This is an especially significant problem when modification (or even recompilation) of the application program is impossible (*e.g.*, commercially distributed software).

Rather than redefining the schema and converting the instances, the second strategy, **emulation**, is based on a **class versioning scheme**. Each class evolution defines a new version of the class and old class definitions persist indefinitely. Instances and applications are associated with a particular version of a class and the runtime system is responsible for simulating the semantics of the new interface on top of instances of the old, or vice versa. Since the former schema is not discarded but retained as an alternate interface, the emulation scheme provides program compatibility. Such a facility has been developed for the Encore system [21].

Encore pays for this additional functionality with a loss in runtime efficiency. Under a conversion scheme, the cost of the evolution is a function of the number of affected

instances. Once converted, an old instance can be referenced at the same cost as a newly-created one. However, the cost of emulation is paid whenever there is a version conflict between the application and a referenced instance.

We feel however that program compatibility among schema versions is a very desirable feature under certain circumstances. It can be of great utility in situations where the database is shared by a variety of applications, as in Computer-Aided Design or Office Automation Systems, when the database acts as a common repository for information, accessed by a variety of applications.

Our scheme supports program compatibility by maintaining multiple versions of the database scheme. Old programs can continue to interact with the database (on both new instances and old) using the former interface. Rather than emulating the evolved semantics all at runtime, efficiency is gained by representing each object as an instance of each version of its class. In this manner, our system effects a compromise between the functionality of emulation and the efficiency of conversion.

Another failing common to the conversion-based evolution facilities is the limitations placed on the variety of schema evolutions that can be performed. Most existing systems restrict admissible evolutions to a predefined list of schema[1] change operations (*e.g.*, adding/deleting an attribute or method from a class, altering a class's inheritance list). The length of this list might vary from system to system, but they are all similar in the way they support change: *The set of changes that can be performed are those which require either a fixed conversion of existing instances or no instance conversion at all.* Unfortunately, change is inherently unpredictable. A desired evolution is sometimes *revolutionary* and under such circumstances, these systems prevent the database programmer from performing the desired changes.

We are interested in supporting evolution in a liberal rather than a conservative fashion; rather than the system offering a list of possible evolutions to the programmer, the programmer should be able to specify arbitrary evolutions and rely on the system for assistance and verification. Change is a natural occurrence in any engineering task, and engineering-support systems should help rather than hinder when an evolution is required.

Although Encore's emulation facility restricts the breadth of class evolution that can be installed, the restrictions are of a different form. Since instances, once created, cannot change their class-version, evolutions that require additional storage for each instance cannot be defined.

[1]Throughout this paper, *schema* refers to the complete collection of class definitions and *class* refers to one a particular type.

In the remainder of this paper, we present a model for specifying schema evolutions and instance adaptation strategies. Our system supports program compatibility, accepts a larger variety of evolutions than existing systems, and supports a variety of options to make it more efficient than the pure emulation facility of Encore.

Previous Schemes

Before describing our system in detail, we present a more detailed description of some important, existing systems. Newer systems, representing related work, are detailed later in the paper (p. 17).

ORION

The most ambitious and effective example of a schema evolution support facility is that provided by ORION [2, 16, 17]. ORION provides a catalogue of schema evolution operations. It also defines a database model in the form of invariants that must be preserved across any valid evolution operation and a set of rules that instruct the system how best to maintain those invariants. Under this model, a schema designer specifies an evolution in terms of the taxonomy, the system verifies the evolution by determining if it is consistent with the invariants and then adjusts the schema and database according to the appropriate rules.

ORION can only perform those evolutions for which it has a rule defined. The set of rules is fixed. For example, changes to the domain of an attribute of a class are restricted to generalizations of that domain. This restriction exists because there is no facility in ORION's evolution language for defining a procedure to convert old instance values. Generalizations of the attribute domain are allowed since this evolution does not require existing instances to be modified.

In ORION, evolutions are performed on a unique schema. Under this scheme, there is no compatibility support for old programs and, depending on the evolution, information contained in the instances might be lost at conversion time. (*e.g.*, deletion of an attribute.)

Encore

Encore implements emulation via user-defined exception handling routines. Whenever there is a version conflict between the program and the referenced instance, the routine associated with that method or instance (and those pair of versions) is called. The routine is expected to make the method's invocation conform to the expectations of the instance or make the return value from the method invocation consistent with the

expectations of the calling program, whichever is appropriate. It is known, however, that certain evolutions cannot be modeled adequately under this scheme. The problem stems from the fact that each object can only instantiate a single version. If an evolution includes the addition (subtraction) of information (*e.g.*, the addition (deletion) of an attribute), there is no place for older (newer) instances to store an associated value. The best a programmer could do in such a system is associate a default attribute value for all instances of older (newer) type-versions by installing an exception handling routine to return the value when an application attempts to reference that attribute from an old (new) instance [21].

The Common Lisp Object System

CLOS [22, 15], while not an OODB system, provides extended support for class evolution nonetheless. As Common Lisp system development is performed in an interactive context, class redefinition is a frequent occurrence. Rather than discard all existing instances, CLOS converts them according to a policy defined by the user. The default policy is to reinitialize attribute values that no longer correspond to the attribute domain and delete attribute slots that are no longer represented in the class definition. Users can override this policy by defining their own method that is called automatically by the system. This method is passed as arguments the old and new slot values, so relationships between deleted and added attributes can be enforced [22, p.859].

Database Model

As a basis for our discussion, we will employ a simplified object-oriented database model.

All objects found in the database are instances of classes. A class is record type of attributes and methods. An attribute is a private, named, typed representation of state, which can be accessed only by class methods. Methods are descriptions of behaviour and can be public or private to the class. Under these restrictions, the set of public methods describe the *interface* of the class, while the attributes model its *state*. Classes are arranged in a type heterarchy (*i.e.*, a directed, acyclic graph): a class's interface must be a generalization of the union of its superclasses (supertypes); the specification of each inherited method must be at least as general as those of its superclasses. (It should also be stated that methods must have semantics consistent with those of their class's supertypes.) Instances of a class that is a declared subtype of another class can be referenced as if they were instances of the superclass.

In addition to the classes, other supported types include primitive types (*e.g.*, integer, floating point number, character) and arrays.

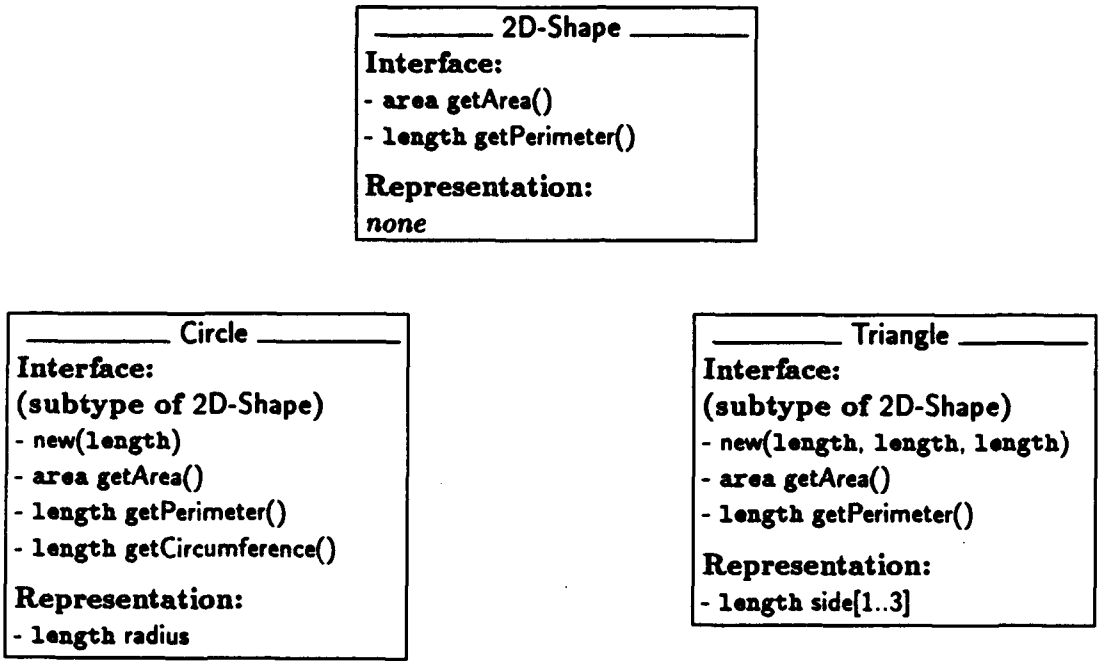


Figure 1: Sample Database Subschema

The set of defined classes comprise the database schema. Each class has an associated unique ID. All objects found in the database are instances of classes. Each instance is identified by a unique *Object ID* (OID), and is tagged with its *Class ID* (CID).

The subsequent section extends this basic model to support class versioning and instance adaptation.

Conversion and Compatibility

Schema Modification vs. Class Versioning

The schema evolution support provided by such systems as ORION and GemStone is restricted to what Kim calls **schema modification**, that is, *the direct modification of a single logical schema* [17]. When only one database schema exists, it is appropriate for the system to convert all existing instances. From a database consistency perspective, it must *appear* that all instances have been converted when the evolution operation is applied.[2] In fact, we would claim that it is the *only* sensible approach.

As has already been stated, however, conversion might render the instances inaccessible to applications that had previously referenced them. The adaptation strategy converts the instances but does not alter procedural references. Thus, application programs written and compiled under the old schema may now be obsolete, unable to access either the old, now converted, instances, or the ones created under the new schema.

A reasonable direction of research here would be to provide some automated mechanisms to assist with program conversion; it is an active line of research [13, 1]. In the OODB context, some work has been conducted at providing support to alert the programmer about the procedural dependencies of their evolution operation [9]. But this is not the only possible solution. Rather than adjust programs to conform to the data, it would seem easier to adjust the data to conform to the existing programs. Also, it is not always possible to alter, or even recompile, programs (*e.g.,* commercially available software). This lack of compatibility support is our primary motivation for adopting a class versioning design for evolution management and support.

Under a class versioning scheme, multiple interfaces to a class, one per version, are defined. When compiled, application programs are associated with a single version of each of the classes it refers to; a *schema configuration*, if you will. With the database populated with instances of multiple versions of a class, the runtime system must resolve discrepancies between the version expected by the application and that of the referenced instance.

Objects Instantiating Multiple Versions

Under the original Encore scheme [21], instances never change their type-version. Aware of the restrictions this causes (*cf.,* previous section), Zdonik proposed a scheme whereby an existing instance can be "wrapped" with extra storage and a new interface, enabling it to be a full-fledged instance of a new type-version [24]. While still accessible through its original interface/version, the wrapped object can also be manipulated through

[2] Whether the instances are converted eagerly or lazily becomes an implementation issue.

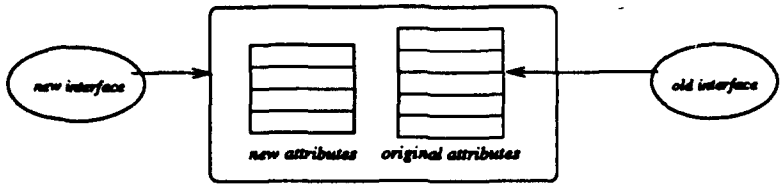


Figure 2: Zdonik's Wrapping Scheme: as in the Encore design, multiple interfaces to the class are preserved. Here, extra space is allocated for the attributes added as a result of the evolution, and applications can access the instance through either the old or new interfaces.

the new interface. Thus, if the class evolution specifies the addition of an attribute, the wrapping mechanism could allocate storage for the new slot in existing instances, without denying backward compatibility. (*cf.*, Figure 2)

Our proposed scheme is a generalization of this approach. Much as each class has multiple versions, each instance is composed of multiple **facets**. Theoretically, each facet encapsulates the state of the instance for a different interface (*i.e.*, version). The representation of these instances is, abstractly, a disjoint union of the representation of each of the versions, and it is useful to consider the representation as exactly that. As will be explained later, however, a wide variety of representations are possible.

As an example, consider a class **Undergraduate**, originally including attributes **Name**, **Program**, and **Class**, and a new version of the class with the attributes **Name**, **Id Number**, **Advisor**, and **Class Year**. (**Class** is one of {Freshman, Sophomore, Junior, Senior}, while **Class Year** is the year the student is expected to graduate.) **Program** is the degree program in which the student is enrolled, and **Advisor** is his academic advisor. While instances of **Undergraduate** in the database will contain all seven distinct attribute slots, any particular application will be restricted to one version and thus only have explicit access to one facet.

In reasoning about the relationship between any two versions of a class,[3] it is useful to divide the attributes into these four groups:

Shared: when an attribute is common to both versions,[4]

[3] For explanatory purposes, imagine that we are describing a class consisting of only two versions, and where the database is populated by instances of both.

[4] Common in the semantic sense, not just having the same name or type.

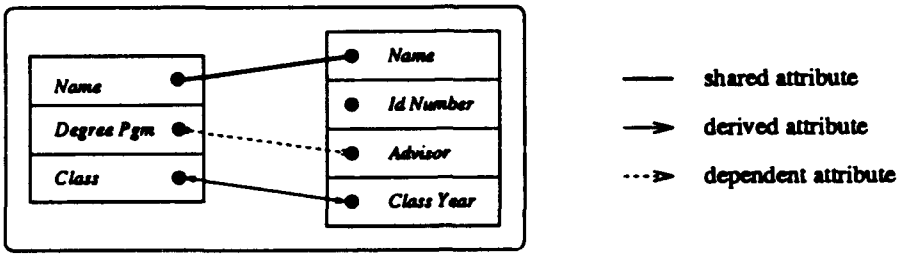


Figure 3: Disjoint union representation of the versioned class **Undergraduate**

Independent: when an attribute's value cannot be affected by any modifications to the attribute values in the other facet.

Derived: when an attribute's value can be derived directly from the values of the attributes in the other facet,

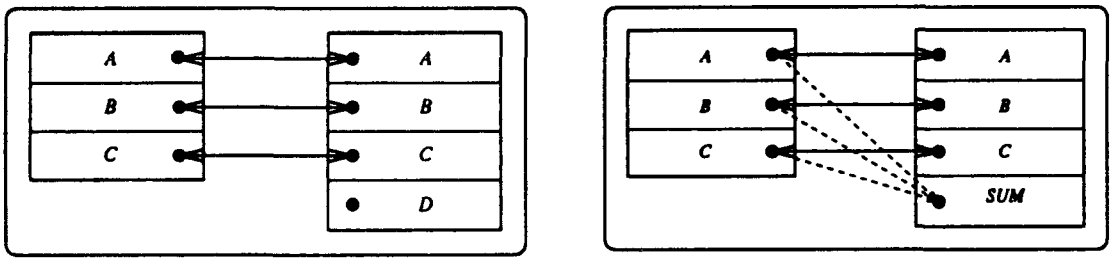
Dependent: when an attribute's value is affected by changes in the values of attributes in the other facet, but cannot be computed solely from those values.

In our example, the **Name** attribute is shared by the versions, while **Id Number** is independent. **Class** and **Class Year** are both derived attributes since, given the current date, it is possible to derive one from the other. **Advisor** is a dependent attribute, since a change in **Program** might necessitate a change in advisor. Likewise, **Program** is a dependent attribute, since a change in advisor might imply that the student has switched degree programs. (*cf.*, Figure 3)

Zdonik *et al.* [21, 24] almost always cite evolutions involving independent or derived attributes in their examples. The original Encore emulation scheme is adequate for supporting evolutions that introduce shared and derived attributes. Zdonik's wrapping proposal addresses the problems associated with independent attributes. Our proposed scheme, however, will provide a mechanism for managing class evolutions that include dependent attributes. (See Table 1 (p.19) for a comparison of the evolution capabilities of various systems.)

Interacting with Multifaceted Instances (Example)

Consider an evolution specification language which can categorize attributes. This is accomplished by associating extra information with each attribute, namely:



(a) (b)

Figure 4: Some simple evolutions: (a) illustrates the relationships following the addition of an independent attribute, (b) shows the addition of a derived attribute.

with **Shared attributes**, associate the name of the corresponding attribute from another version,

with **Derived attributes** — a function for determining its value in terms of the values of the attributes in the other version, and

with **Dependent attributes** — a function for determining its value in terms of the values of the attributes in both versions (*i.e.*, the entire object), and

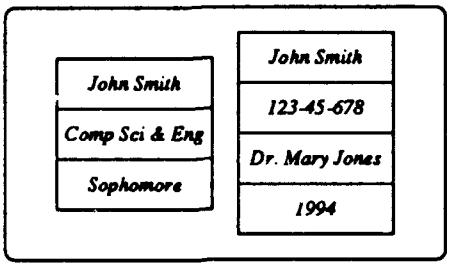
with **Independent attributes** — nothing extra at all.

Representing the class instances as a disjoint union of the version facets, as described earlier, consistency between the facets can be maintained according to the following procedure:

> *Whenever an attribute value of a facet is modified, those attributes in the other facet that depend on it must be updated. For shared attributes, the new value is copied; for dependent and derived attributes, the dependency functions are applied and the result written into the (attribute) slot in the other facet.*

The remainder of this section consists of an example:

Consider a multifaceted instance of the **Undergraduate** versioned class, represented graphically as follows:



Imagine that John Smith returns to university after his first summer vacation and wishes to change to the undergraduate Math program. Also, he had taken some summer classes that have given him enough credits to graduate a year early. The change to his data record are recorded through an application program employing the first version of the **Undergraduate** class. The system must now propagate those modifications to the second facet.

The attributes Class and Class Year can be derived in terms of each of other. A reasonable derivation function for Class Year is:

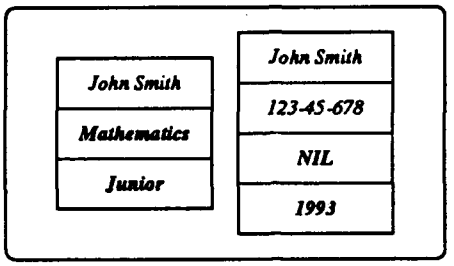
$$\text{ClassYear} = \begin{cases} \text{cy} + 3 & \text{if Year} = \texttt{Freshman} \\ \text{cy} + 2 & \text{if Year} = \texttt{Sophomore} \\ \text{cy} + 1 & \text{if Year} = \texttt{Junior} \\ \text{cy} & \text{if Year} = \texttt{Senior} \end{cases}$$

Where cy is the current year. The Advisor attribute is dependent upon the value of the Program attribute, but not completely derivable. A reasonable dependency function is:

$$\text{Advisor} = \begin{cases} \text{Advisor} & \text{if Advisor} \in \text{Program faculty} \\ \text{nil} & \text{otherwise} \end{cases}$$

Since there is not enough information to derive it, the student's advisor will have to be filled in later.

Applying these functions in concert with the desired changes to John Smith's record, the multifaceted instance becomes:



Representing Multifaceted Instances

In the previous section we described the semantics of our schema versioning scheme. In this section we address the issue of how to realize these multifaceted instances physically in the database.

Our basis for consideration is a system which implements the design as described: class evolutions are defined by creating a new version of the class; a new facet (corresponding to the new version) is associated with each instance of the class and initialized according to a user-defined procedure. Each application program interacts with the instances through a single version (interface) and modifications to attribute slots on the "visible" facet are immediately propagated to the other facets, using a mechanism similar to the trigger facility found in many relational and AI database systems [23, 12].

The most obvious target for improvement in this scheme is how new facets are added. The allocation and initialization of new facets for existing instances at evolution time can be deferred until such time as the facets are actually needed (*i.e.*, by an application). In this way, some of the runtime and most of the space costs of supporting multiple versions are only spent when absolutely necessary.

The strategy of deferring the actual maintenance of a dependency constraint until its effect is actually required can be applied as well to the propagation of information among the facets of an instance. Rather than update the attribute values of the other facet(s) each time a facet attribute is modified, one need only bring a facet up-to-date when there is an attempt to access it. This scheme can be supported by associating a flag with each facet indicating whether the facet is up-to-date with respect to the most recently modified facet. The application of read methods on facets with an unset flag are preceded by a resynchronization operation, which performs any necessary updates and sets the flag.

This scheme reduces overall runtime expense, since the resynchronization step is not performed in concert with every update operation, as was previously the case. However, it does increase the potential cost of previously inexpensive read operations.

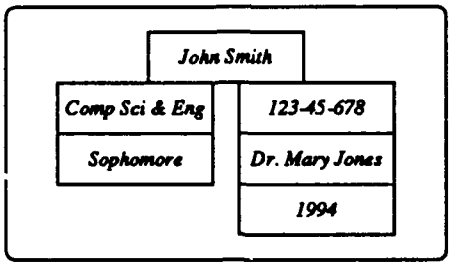


Figure 5: Multifaceted instance representation using common slot for shared attributes

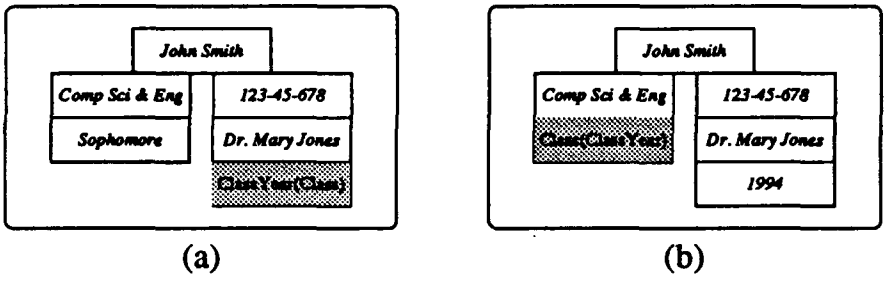
To this point, we have been very liberal with our allocation of space for instance representation. Although the lazy allocation of facets conserves some space in the short run, the disjoint union representation model implies that every instance of a versioned class will have a complete collection of facets. There are a few optimizations that could be performed to reduce space requirements.

The first space-saving improvement entails having each set of shared attributes occupy a single slot in the multifaceted representation. A performance improvement might also be realized here, since slot sharing reduces the expense and/or frequency of update propagations. (*cf.*, Figure 5 (p.14).)

Under certain circumstances, the slot associated with a derived attribute can be recovered as well. If an inverse procedure to the derivation function is known to the system, then the attribute can be simulated by appropriate reader and writer methods. For many evolutions, the inverse procedure appears as the derivation function for the related attribute in the other facet. The Year and Class Year attributes in our example (p.9) are related in that way. (*cf.*, Figure 6 (p.15).)

From a runtime performance perspective, this space optimization reduces the expense of write methods while making read methods more costly. The slot allocated for a derived attribute acts as a cache for its derivation function and, depending on the frequency of modifications to its dependent attributes in the other facet(s), its maintenance might be more time-efficient.

In fact, the emulation scheme of the Encore system is an extreme case on the space vs. time spectrum. In the Encore system, however, emulation was the only option. In our scheme, the programmer could choose to completely emulate a facet in situations where time is less of a concern than space, and where all the attributes are derivable from other facets.



(a) (b)

Figure 6: Multifaceted instance representation minimizing derived attribute allocation. For the **Undergraduate** class, two minimizations exist.

Forcing an Adaptation Strategy

While important in general, program compatibility is not always required (*e.g.*, a database with a single application program and a single user). In such situations one should be able to employ the minimally expensive strategy and instruct the system to convert existing instances fully and discard (or perhaps archive) the old information. Furthermore, conversion and compatibility are not mutually exclusive. As long as an inverse conversion procedure is known, one could convert and emulate the older interface. This might be useful when you want to preserve compatibility, but expect that it will be needed infrequently enough that you are willing to pay the cost of emulation in those instances. If an application tends to reference a distinct subset of the instance collection, one could employ a strategy that converts (on access) instances to the version of the application.[5] The importance characteristic of this evolution architecture is that the database programmer has access to the control knobs and can tune the evolution strategy to improve performance.

Sometimes, modification of the database or its schema is impossible. Databases might be read-only for permission (*e.g.*, remote database exported as a public service) or licensing reasons (*e.g.*, reference materials on CD-ROM). In such situations, something resembling Zdonik's wrapping scheme (*cf.*, p.8) must be used, with the wrapper actually residing in a separate database.

Subtyping

We have, to this point, failed to explain how our class evolution and instance adaptation scheme copes with our model's subtyping mechanism. We first identify what

[5]This is the approach taken by Monk's CLOSQL system [19]. See p.18 for details.

characteristics of subtyping are problematic with respect to evolution and adaptation and then motivate our solution.

Evolutions relating to subtyping can be divided into two categories: changes to the members of a class's set of supertypes, and evolutions to the superclasses (supertypes) themselves. The former present no special problem: while the addition of a new supertype might necessitate changes to the class's interface, these changes are no different from independent evolutions on method specifications.

Unfortunately, support for evolutions upon a superclass (*i.e.*, a class with declared subtypes) is problematic. Our basic database model specifies the following:

> *A class's interface must be a generalization of the union of its superclasses (supertypes); the specification of each inherited method must be at least as general as those of its superclasses.*

The essence of the problem lies in the fact that a class could evolve in such a way as to interfere with its subtypes. As an illustrative example, reconsider the subschema example depicted in Figure 1 (p.7), where two classes, Circle and Triangle, are declared to be subtypes of a third class, 2D-Shape. Consider the following two evolutions on 2D-Shape: 1) where the public method, getPerimeter, is removed, and 2) where a new method, draw, is added to the interface.

The removal of getPerimeter from the interface does not affect the subtype classes, which can continue to export getPerimeter in their interfaces and maintain their subtype relationship with the new version of the class 2D-Shape. With the introduction of the method draw to the 2D-Shape interface, however, the Circle and Triangle classes cannot (without being evolved themselves) be subtypes classes to the new version of 2D-Shape.

Note that not all method additions cause this problem: if one were to reintroduce getPerimeter to the interface of 2D-Shape, the subtype classes, which are already exporting a compatible method (*i.e.*, their signatures are a supertype of the signature of the corresponding method from the supertype class), could remain subtype classes to the new version.

The problem introduced by the addition of methods to superclasses has been addressed in previous work. The ORION system disallowed evolutions that upset the type graph [2]. Zicari has proposed a transaction model, which would allow the programmer to evolve the class and its subtype classes in one atomic, evolution operation [26, 25]. Under our liberal evolution scheme, a third option is possible: the versioning of the type heterarchy.

With the versioning of the type heterarchy, type membership would depend on the version of the superclass being referenced: following the addition of an incompatible public method to a superclass, instances of the declared subtypes would be accessible

as instances of the old version of the superclass, but not as instances of the new version. Class versioning was introduced in our scheme so as to allow instances to remain accessible in across evolutions to their constituent classes, but the versioning of the type heterarchy compromises compatibility.

The best (*i.e.*, the most practical) approach would be to evolve the entire subtype graph when evolutions to the root class would "disown" its subtype classes. However, just as programmers are free to not support compatibility, they should have the freedom to version the type graph, if they feel it appropriate.

Compound Evolutions and Version Configurations

Evolutions to supertype classes motivate support for the atomic evolution of multiple classes. Zicari proposed such a mechanism to support compound evolutions on classes (a primitive set of evolution operations was supplied, forming a basis for all supported evolutions) [26, 25]. A similar system can be employed here to support possible evolutions on type subgraphs, as well as some other compound evolutions, such as the *telescope* evolution, described in detail below (p. 20).

Just as an instance adaptation strategy is required for each evolution, a corresponding adaptation construct for evolution transactions is necessary. This construct, which we call a *version configuration*, maintains the version dependencies among the evolved classes. It would contain the knowledge that a particular subclass instance needs to be adapted to a particular version before it could be accessed as an instance of a particular superclass (supertype) version.

Related Work

There are a number of other research projects currently concerned with evolution in the OODB context.

Bertino [4] presents a schema evolution language which is an OODB adaptation of the view mechanism found in many relational database systems. Her primary innovations are the support of inheritance and *object IDs* (OIDs) for view instances, two important characteristics of OODB models that are not present in the relational model. View instances with OIDs are physically realized in the database, enabling the view mechanism to support evolutions that specify the addition of an attribute, as envisioned by Zdonik [24]. However, Bertino's scheme focuses on how evolutions affect the schema. It is not concerned explicitly with the effects upon the instances nor with compatibility issues.

Zicari proposed [26, 25] a sophisticated evolution facility, providing an advisory program to determine at evolution time whether the evolution is consistent with inter-class and method dependencies. Evolution transactions are introduced to allow for compound evolution operations. However, Zicari's lack of concern for instance adaptation is evident; by defining the attribute-renaming evolution as the atomic composition of the attribute-delete and attribute-add operations, his scheme fails at the instance level.

Monk's CLOSQL [19] implements an class versioning scheme, but employs a conversion adaptation strategy. Instances are converted when there is a version conflict, but unlike ORION and GemStone, CLOSQL can convert instances to older versions of the class if necessary.

Lerner's OTGen design [18] addresses the problem of complex evolutions requiring major structural conversions of the database (*e.g.*, information moving between classes, sharing of data using pointers) using a special-purpose language to specify instance conversion procedures. As it was developed in an integrated database context, where the entire application set is recompiled whenever the schema changes, versioning and compatibility were not considered. However, Lerner's language supports a variety of evolutions and associated adaptations that are not addressed in many other papers, most notably evolutions that alter the structure of shared component objects.

Bratsberg [5] has been developing a unified semantic model of evolution for object-oriented systems. Similar to our work, compatibility for old clients is described in the context of relations, maintaining consistency between views.

One significant difference between our respective threads of research is our concentration on the variety of adaptation strategies and representations for the (possibly) multifaceted instances. This is reflected in this paper's discussion of the range of possible adaptation strategies, depending on the (expected) access patterns of the affected instances.

In addition, Bratsberg introduces an alternative consistency maintenance scheme, operating at the method rather than the representation level. Such a scheme might be more suitable in situations where the native representations of the various versions are drastically different or when the relations among the various representation attributes is hard to define.

Description of Evolution	ORION	Encore	Bertino[4]	Us
Add/delete/rename a method	√	√	√	√
Add an attribute	√	×	√	√
Delete an attribute	√	√	√	√
Generalize the domain of an attribute	√	√	√[a]	√
Change (arbitrarily) the domain of an attribute	×	√	√[a]	√
Telescope[b]	√	√	√	×[c]
Change supertypes	√	?	√	√

Table 1: Some evolutions and which systems support them. Note that none of the other systems directly address instance adaptation.

[a] This evolution cannot be defined directly, but can be implemented by replacing the attribute in the new version with generic methods for reading and writing.[3]

[b] See p. 20 for definition.

[c] Support for telescope evolutions is not possible in our database model, as currently presented. (*cf.*, p. 20 for discussion.)

Ongoing Work

A number of shortcomings of my system have been mentioned above. This section discusses a few of them in more detail, and outlines some possible approaches for their removal.

Data Model Extensions and New Evolutions

Our basic model was designed for demonstrative purposes; while providing an adequately powerful model for our discussion of schema evolution and instance adaptation, it lacks some of the features characteristic of many real-world OODBMS and thus fails to address certain evolutions that occur under such systems. We describe how our design could be extended to cope with some of the other evolutions identified in previous works, and the problems these extensions introduce.

Pointer references and Telescope Evolutions

Our database model can be extended to allow for pointer references, so that objects can be multiply-referenced. (Objects could already be embedded into other objects in our model, by typing a class attribute to another class.) With the introduction of pointer references, however, *telescope* evolutions, becomes troublesome to support.

A **telescope** evolution involves the simultaneous modification to the representation of two classes: an attribute of one class is moved out of a class to become an attribute (or set of attributes) of a dependent class [20, 4]. (An example is presented in Figure 7 (p.21).) The reverse operation, termed *aggregate* [20] sees a set of attributes moved "down" into a component class.

Similar to the renaming of an attribute, the telescope evolution cannot be simulated by attribute-delete and attribute-add operations on the appropriate classes, because we are concerned with preserving the values of the telescoped attribute at instance adaptation time.

Note that telescoping is a compound, *inter-modular* evolution. In such a situation, it is reasonable to expect the adaptation routine to be inter-modular as well, requiring access to the actual attributes of class instances, and not just its public methods.

First, let us imagine that compatibility is not a concern; that we are solely interested in converting instance-pairs of the old class definitions to pairs of the new definitions. In this case, we observe that both objects (*i.e.*, the referring object and the object being referred to) must be converted simultaneously; otherwise the value of the telescoped attribute would be lost. The *version configuration* facility described earlier (p.17) is insufficient to the task because, in order to perform the adaptation, the (new) object

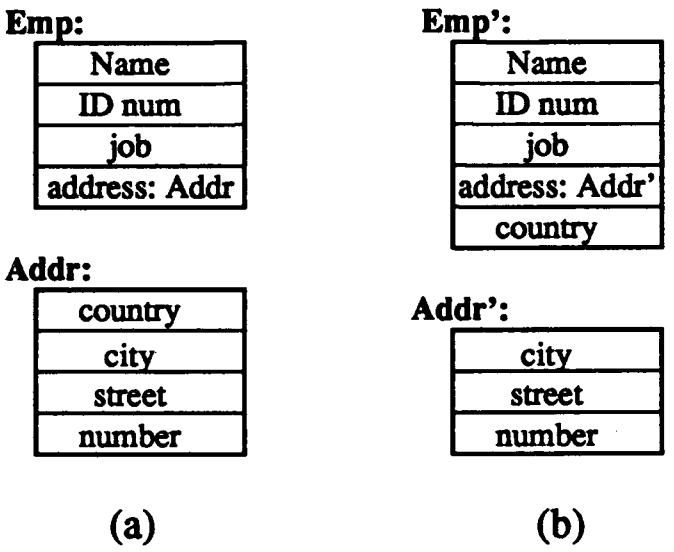


Figure 7: Example of a *telescope* evolution: The country attribute of the Addr class is moved "up" into the Emp class. Figure (a) illustrates the original schema, Figure (b) the evolved one. Note that from the point of view of the Addr Class, the evolution resembles an attribute deletion.

gaining the attribute requires access to the attribute found in the (old) object, before it is discarded. Another more problematic, issue concerns multiply-referenced objects — a common occurrence, since information sharing is probably why the pointer reference was employed in the first place. If an object is converted, so must *all* objects that refer to it.

Ironically, these problems are simplified somewhat if we are willing to provide compatibility support. We do not have to worry about the value of the telescoped attribute being lost, because it will continue to be stored in the old facet of the object. Likewise, we do not need to adapt all objects referring to an adapting instance, since the referring objects can continue to access the adapted object through its former interface — just like older client applications. It should be observed that in moving an attribute "up" from the component to the referring class, backward compatibility would necessitate the addition of a ***back pointer*** (from the component to each referring object). (See Figure 8 (p.22) for an illustration.) This back-pointer is similar in purpose to the **connection** relationship introduced by Bertino [4, p. 148]. (For the opposite evolution, *i.e.*, *aggregate*, a similar pointer would be needed for forward compatibility purposes.)

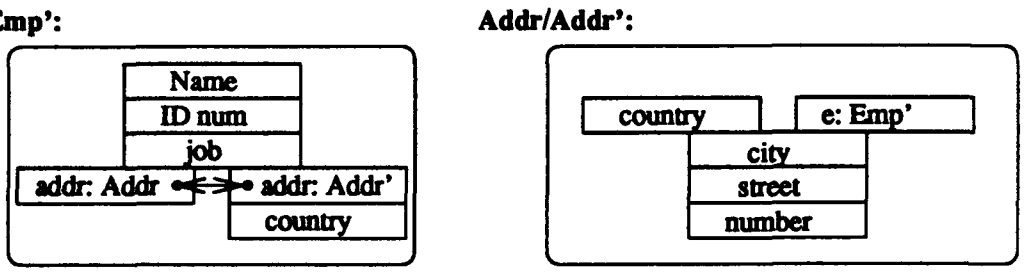


Figure 8: Data representation for compatibility support in the presence of a telescope evolution. A pointer back from Addr to Emp is required for compatibility purposes.

Unfortunately, a serious problem remains. The multi-faceted representations of the various objects involved will preserve multiple copies of the telescoped attribute, copies that must remain consistent with each other. Our existing model provides for the consistency maintenance among the facets of an object, but not *across* objects. Future research will determine how to enhance our model to cope with this problem.

Inheritance

By inheritance, we refer to the ability of a subclass to inherit the representation of its superclass, in addition to the external interface. Such mechanisms have been characteristic of many object-oriented languages and OODB systems. The difficulties in integrating such a mechanism with our evolution framework stems from the way the modularity barrier is broken; a subclass has access to the internal state it inherited from its superclasses. Thus, evolutions involving the representation of a class affect all its subclasses as well. For such evolutions, both classes should be evolved simultaneously, as has been discussed before in the context of subtyping. (p.17)

It has been noted elsewhere that specialization (*i.e.*, subclassing) is nothing but one type of evolution[7, 5]. Within our evolution scheme, subclassing, might not be required as an independent mechanism; its features are likely supportable by the adaptation system. (See Figure 9 (p.23) for an example.) This issue will be investigated in a future paper.

Merging Two Arbitrary Classes

Another interesting evolution is the merging of two arbitrary classes into one. Such evolutions are feasible provided that there is a one-to-one relationship between the

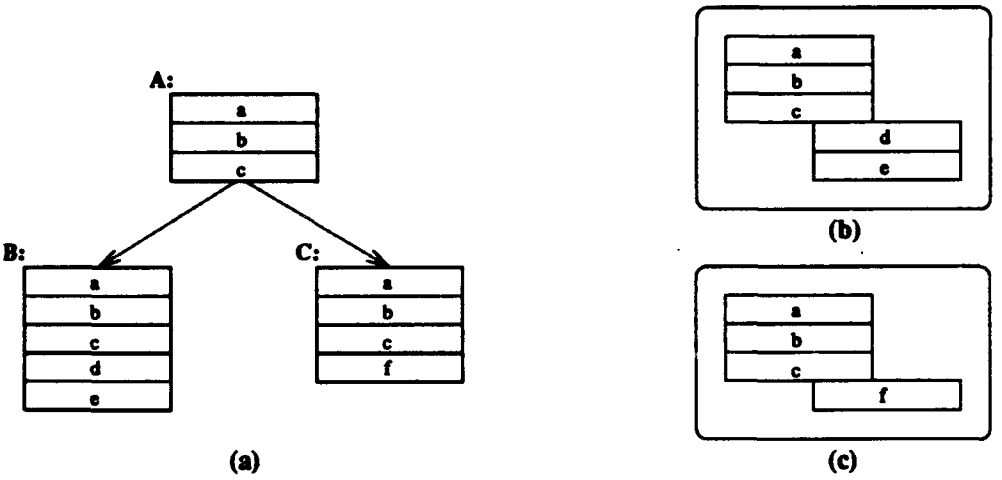


Figure 9: Subclassing as a Specialization Evolution: The standard subclassing mechanism could be implemented using a restricted form of class versioning and instance adaptation. Figure (a) depicts three classes: B and C as subclasses of A. Figure (b) presents Class B as a new version of A. Figure (c) does likewise for C. Like the formal subclasses, the versions depicted in (b) and (c) can substitute for (*i.e.*, are compatible with) instances of A.

instances of the affected classes. For backward compatibility, the resulting class should be able to emulate the features of the two classes it was evolved from, much the way a subclass emulates its multiple superclasses in a system supporting multiple inheritance.

Programming an Adaptation Strategy

Our system as described has more versatility than ORION's facility because it supports user-defined instance adaptation information. Consistent with our desire to aid the schema designer, we would like to provide the ability to install user-defined adaptation strategies based on disjoint union data model. A few examples include:

Offline conversions: For example, a database that must be highly available during business hours could maintain a log of the instances touched during the day and spend the idle overnight hours converting them. If the economics warrant it, instances could monitor their usage patterns and adapt their representations to their applications.

Online optimizations: Instead of converting all instances to a single representation, the system could employ different representations, depending on the access pattern to the object. For instance, in a situation where one of several existing interfaces predominates for a time period, the following strategy might serve well: maintain only one consistent facet at a time, the one corresponding to the version interface that was used most frequently in the recent past. (For example, maintain facet A, until such time as the B interface is used three times consecutively, at which point begin maintaining facet B.)

If the space cost on maintaining multiple facets is prohibitive, then the programmer has to struggle between emulation and converting strategies. The optimal solution (ignoring administrative costs) would be to emulate the new interface until it becomes clear that conversion would be more efficient. A family of online algorithms working on this premise are possible. However, all require a cost model for the conversion and emulation strategies, which is being developed by the author.

Elaborations on such schemes will be a subject of another paper.

Conclusions and Summary

In this paper, we have described a specification model for schema evolution that has the following features:

- Schema versioning instead of modification to a single schema, so that program compatibility can be supported, if desired. Schema versioning also permits multiple views of the database to co-exist indefinitely.
- Compatibility support is provided at less runtime cost than the Encore facility. For each version of a class, each instance has a corresponding facet. For attributes which can be derived solely from attributes from other facets, this facility is like a cache, sacrificing space for time. For attributes which are not reflected in the representation of the versions, the facet provides space for the value to stored, thereby preserving information that would be lost under a conversion scheme.
- A broader variety of evolutions are supported than in existing systems (ORION, GemStone, Encore). All evolutions involving changes to a single class are supported, and future extensions have been described that will add support for some more complex evolutions (*e.g.*, telescoping).
- It is possible to evolve a class's interface without consideration to the classes which inherit it (*i.e.*, *subtypes* in our paper), but with some unfortunate results. It would be better to evolve the subclasses in tandem with their parent.
- Tuning of the adaptation scheme by the programmer is possible, allowing the programmer to decide how much duplication of information is present in each (multifaceted) instance. If one version is used very infrequently, the programmer can save space and time, by emulating its interface instead. And, as has already been mentioned, compatibility need not be supported at all if it is not desired.

Acknowledgements

I am grateful to James H. Morris, and especially to Michael Horowitz, for the relevant comments and criticisms they provided during the writing of this paper.

References

[1] Arnold, R. S. **Tutorial on Software Restructuring.** Institute of Electrical and Electronic Engineers. IEEE Society Press, Washington, DC, 1986.

[2] Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases.* in: **Proceedings of the SIGMOD International Conference on Management of Data,** edited by U. Dayal and I. Traiger. San Francisco, CA, 1987.

[3] Bertino, E. 1992. *Personal Communication.*

[4] Bertino, E. *A View Mechanism for Object-Oriented Databases.* in: **Advances in Database Technology – EDBT '92 International Conference on Extending Database Technology,** edited by . Vienna, Austria, 1992, pp. 136–151.

[5] Bratsberg, S. E. *Unified Class Evolution by Object-Oriented Views.* in: **Proceedings of the 11th International Conference on the Entity-Relationship Approach.** 1992. *to appear.*

[6] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., and Williams, M. *The GemStone Data Management System.* in: **Object-Oriented Concepts, Databases and Applications,** edited by W. Kim and F. H. Lochovsky. Addison-Wesley, Reading, MA, 1989.

[7] Casais, E. *Managing Class Evolution in Object-Oriented Systems.* in: **Object Management,** edited by D. C. Tsichritzis. Universite de Geneve, Centre Universitaire d'Informatique, Geneva, 1990, pp. 133–195.

[8] Casais, E. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach,* ScD. Thesis. Université de Genève, Geneva, 1991.

[9] Delcourt, C. and Zicari, R. *The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System.* in: **Proccedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science,** vol. 512, Springer-Verlag, Geneva, Switzerland, 1991. *A more detailed version is available as [10].*

[10] Delcourt, C. and Zicari, R. *The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System.* Dipartimento di Elettronica Technical Report, no. 91.021, Politecnico di Milano, Milan, Italy, 1991. *A short version of this paper appears in the 1991 ECOOP proceedings.*